The Sync Model:
A Parallel Execution Method for Logic Programming

Pey-yun Peggy Li
and
Alain J Martin

Computer Science Department
California Institute of Technology

5221:TR:86

The Sync Model:

A Parallel Execution Method for Logic Programming

Pey-yun Peggy Li
and
Alain J. Martin
Computer Science
California Institute of Technology
Pasadena CA 91125

March 1986

5221:TR:86

Abstract

The Sync Model, a parallel execution method for logic programming, is proposed. The Sync Model is a multiple-solution data-driven model that realizes AND-parallelism and OR-parallelism in a logic program assuming a message-passing multiprocessor system. AND parallelism is implemented by constructing a dynamic data flow graph of the literals in the clause body with an ordering algorithm. OR parallelism is achieved by adding special Synchronization signals to the stream of partial solutions and synchronizing the multiple streams with a merge algorithm. The ordering algorithm and the merge algorithm are described. The merge algorithm is proved to be correct and therefore, the Sync Model is proved complete, i.e., the execution of a logic program under the Sync Model generates all the solutions.

The research described in this paper was sponsored by the Defense Advanced Research Projects Adency, ARPA Order No. 3771, and monitored by the Office of Naval Research under contract number N00014-79-C-0597.

1 Introduction

One way to improve the efficiency in the execution of a logic program is to exploit the potential parallelism, namely AND parallelism and OR parallelism, inherent to the program. In this paper, a method – called the "Sync Model" – is proposed for the parallel execution of logic programs on a message-passing multiprocessor system. The method realized both AND-parallelism and OR-parallelism. OR parallelism – the parallel execution of all clauses that are unifiable with the goal – is easier to realize than AND parallelism because the executions of OR clauses are independent of each other. On a message-passing system, the synchronization of the multiple solutions generated by different processes is the major problem in the implementation of OR parallelism. AND parallelism—the parallel execution of AND literals in a clause body—may result in binding conflicts for a variable shared by several literals.

Constructing a data flow graph is the most common approach for AND parallelism. By allowing exactly one producer for each shared variable, binding conflicts can be eliminated. One problem in the data flow approach is that the data flow graph is changed dynamically according to the binding values transmitted within the graph. When a variable is bound to a partially instantiated term containing another variable, binding conflicts may occur. Therefore, the data flow graph needs to be modified to enforce the "one producer per variable" rule to the new variable. In most computation models for concurrent logic programming languages, the data flow graph of literals in a clause body is constructed by the programmer through variable annotations. Alternatively, the data flow graph can be constructed automatically by the system; either dynamically such as in Conery's AND/OR process model [2] or statically such as in Chang and DeGroot's static data dependency analysis [1,3]. In the Sync Model presented in this paper, the data flow graph is dynamically constructed after each unification and is modified by adding "dynamic links" when partially instantiated terms are detected in a binding by using a run-time type checking algorithm similar to [3]. The algorithm is more efficient than [2] and the graph constructed by the algorithm reveals more parallelism than [1]. Optional variable annotations from the programmer may help constructing the data flow graph.

To implement both AND parallelism and OR parallelism in one model is a difficult task. The synchronization of partial solution streams in AND processes has never been solved satisfactorily. Either AND parallelism is suppressed by connecting sibling AND processes into a linear chain [7, 9] or OR parallelism is reduced by using backtracking [2]. In the Sync Model, a synchronization mechanism is proposed to synchronize the multiple partial solutions so that all the solutions of a

given problem will be produced without explicit request. Therefore, the Sync Model is a multiple solution data driven model.

The language we choose for the Sync Model is an extended logic programming language, called CLP, with optional variable annotations and a commit operator. Variable annotations, the input annotation ("?") and the output annotation ("!"), are used in the clause body to specify the producer and the consumer of a shared variable. The commit operator "→" is used to serialize the executions of two parts of the clause body. CLP is not designed as a concurrent language. The variable annotations and the commit operator are used to achieve more efficient execution under the Sync Model, but they are not required and do not change the semantics of the language. Therefore, although the Sync Model is designed for CLP, any Horn-clause logic program can be executed under the Sync Model.

The target machine for the Sync Model is a message-passing multiprocessor system with the processors interconnected into an augmented binary tree, called the *Sneptree* [8,5]. Since the mapping of an unbounded binary tree onto the Sneptree is done automatically and the mapping of a complete binary tree onto the Sneptree is always optimal, the Sneptree is an ideal architecture for the Sync Model.

One of the major distinction between the Sync Model and the computation models for other concurrent logic programming languages, such as Concurrent Prolog [13], is that in our Model, a process is suspended when waiting for an input from an input channel, while in Concurrent Prolog, a process is suspended when it attempts to unify a read-only variable with a non-variable term. In our approach, all the input variables are bounded before the unification so that the unification rule is not changed. In Concurrent Prolog and other similar approaches [11,12], the unification rules are modified to handle variable annotations. As a consequence, the variable annotations may be propagated to other non-annotated variables and a read-only variable may get instantiated in a unification.

The rest of the paper is organized as follows: In the next section, the language and the Sync Model are described. We also address, and propose solutions to, the main problem of constructing the data flow graph, i.e., binding to a partially instantiated term causes the data flow graph to be changed, as well as the synchronization problem of multiple partial solutions in the data flow graph. In sections 4 and 5, the two main algorithms of the Sync Model, i.e., the ordering algorithm and the merge algorithm, are presented. We also prove the correctness and completeness of the

merge algorithm and the Sync Model.

2 The Language and the Sync Model

2.1 The Language

The language, called CLP, (which stands for Concurrent Logic Programming), is an extended logic programming language with variable annotations and guarded clauses.

A CLP program is a finite set of *guarded clauses* of the form

$$A :- G_1, G_2, \ldots, G_m \rightarrow B_1, B_2, \ldots, B_n.$$

where A is called the *head* of the clause, $(G_1, \ldots, G_m)$ the *guard* of the clause, and $(B_1, \ldots, B_n)$ the *body*.

The guard of a clause may be empty. When the guard is empty, the commit operator is neglected. When both the guard and the body are empty, the clause is called a *unit clause*. Both the guard and the body are a set of literals. The two sets are separated by a commit operator, "$\rightarrow$". Declaratively, the commit operator reads like a conjunction: A is true if $G_1, \ldots,$ and G_m, as well as $B_1, \ldots,$ and B_n are true. Operationally, the commit operator forces the sequential execution of the guard and the body: a goal $A1$ which is unifiable with A can be reduced to $B_1, \ldots,$ and B_n if and only if the guard literals $G_1 \ldots G_m$ are evaluated to true.

A variable can be either a simple variable, or an *output variable* annotated by a postfix operator "!", or an *input variable* annotated by a postfix operator "?". Variable annotations are not allowed in the clause head. This restriction prohibits annotated variables from appearing in the unification. Therefore, Robinson's unification algorithm can be used directly without any modification. A variable is "shared" when it appears in more than one literal in the body. For a shared variable in the body, at most one literal containing that variable is allowed to have it annotated as output. Such a literal is called the *producer* of that variable, and the literals that contain input variables are called the *consumers* of those variables. The guard may not have any shared variable with the clause head or the body after unification — a guard evaluates to true or false without generating any outputs. But share variables between guard literals are allowed. Such a syntactic restriction separates the guard and the body into two independent parts which simplifies the implementation of our Model. In each CLP program, there is a goal with the form "$:- G$".

Unlike other parallel logic programming languages, the extra language constructs in CLP, the variable annotations and the commit operator, do not affect the semantics of the language. They can be used by the programmer optionally to achieve more efficient execution under the Sync Model. In order to prevent the semantics from being changed by the commit operator, when the restriction on the variables of the guard is violated, the system simply ignores the commit operator and executes the guard and the body in parallel.

The execution of a logic program is to construct and search the AND/OR tree of this program. For a given goal and a program, there exists a unique AND/OR tree which represents the complete search space of the goal. The Sync Model constructs a tree of processes corresponding to the AND/OR tree of the program and search the tree in breadth-first manner.

2.2 The Sync Model

The computation model of CLP, called the Sync Model, is a *process model.* Two types of processes are created and terminated dynamically during the computation. An *AND process* corresponds to a goal, and an *OR process* corresponds to a clause that is used to reduce a specific goal. A tree of interleaved AND and OR processes, called the *process tree*, is constructed corresponding to the AND/OR tree of the program. The initial goal is assigned to an AND process, which becomes the root of the process tree. For each clause whose head is unifiable with the goal of an AND process, one OR process is spawned to carry out the unification and the reduction of this OR clause. After unification succeeds in an OR process, the reduction of the goal is carried out by spawning one AND process for each literal in the body and then reducing the goals in the AND processes concurrently. If the clause in an OR process has a nonempty guard, a set of AND processes is spawned for each goal in the guard first. When all the AND processes for the guard successfully terminate, the OR process can spawn processes for the goals in the body and proceed. When any of the guard literals fails, the OR process fails. Therefore, full OR Parallelism is implemented in this model in the way of parallel unification of all the unifiable clauses, parallel evaluation of all the guard literals and parallel execution of all the OR branches that succeed in unification.

A leaf node of the process tree is either an OR process which fails to unify, or an OR process corresponding to a unit clause, or an AND process corresponding to a built-in predicate. An OR process containing a unit clause returns the variable bindings to its father AND process and terminates if it succeeds in unification. An AND process corresponding to a built-in predicate evaluates the predicate directly and sends the variable bindings to proper destination processes.

A non-leaf AND process succeeds when at least one of its OR descendants succeeds. It receives the bindings of its output variables from the descendants and sends them out to its father and all the sibling consumer processes of its output variables. A non-leaf OR process succeeds when all its descendant AND processes successfully terminate. It merges the results received from its descendants and then sends them to its father.

AND parallelism is implemented by dynamically constructing the data flow graph of the literals in the clause body. To avoid binding conflict in the parallel execution of sibling AND processes with shared variables, only one AND process is allowed to be the producer of a shared variable. All the other AND processes that also contain that shared variable are considered the *consumers* of that variable. A consumer process will suspend its computation until the values of its input variables have been received from their producers. A data flow graph of all the literals in the clause body, (so-called AND literals), is constructed such that a node represents an AND literal and an edge is directed from the producer of a shared variable to a consumer of that variable. As we shall see, the ordering algorithm will guarantee that the data flow graph is acyclic so as to avoid deadlock. Communication channels are added into the process tree to model the edges of the data flow graph. With the communication channels between sibling AND processes, the process tree is no longer a tree. We prove later that our process tree generates the same results as the corresponding AND/OR tree.

The input and output annotations in CLP are added to the program optionally by the programmer to help construct the data flow graph so that more efficient computation can be achieved. Without explicit variable annotations, the "left to right" order of the AND literals is used for selecting the producer of a variable. The explicit variable annotation should fulfill the two restrictions on the data flow graph: one producer per variable and acyclicity of the data flow graph. These can easily be checked syntactically.

Parallel execution of different OR processes may produce multiple solutions for the output variables of their father AND process. Those multiple solutions will be transmitted along the communication channels. Hence, we need some mechanism to synchronize the multiple inputs of a given AND process originating from different sources. In our computation model, any process that generates or collects a solution transmits the solution without requiring a request. Hence, our model can be viewed as a *multiple-solution data-driven model.* With this synchronization mechanism, we are able to incorporate both AND parallelism and OR parallelism without any form of backtracking.

2.2.1 Synchronization of Multiple Inputs of a Process

Multiple solutions for a variable may be transmitted through the communication channels in the data flow graph. If one AND process, say p, consumes two inputs from two different sources, we shall merge the two input streams to form all the input combinations of process p. Usually, the input combination of process p is the Cartesian Product of the two input streams. There is one exception — when the two input streams originate in the same process, the input combination of p is a set of Cartesian Products over certain portions of the two input streams that derive from the same output of the common ancestor. In the sequel, we call a set of paths that have the same starting process and the same ending process a *multiple path* between these two processes. In Figure 1, there are two paths (a, b, d) and (a, c, d) between process a and process d. If process a binds (X,Y) to (x_1, y_1) and (x_2, y_2), process b binds T to t_1 and t_2 with input x_1, t_3 with input x_2, and process c binds S to s_1 and s_2 with input y_1, s_3 and s_4 with input y_2, then the input combination for process d should be $(t_1, s_1), (t_1, s_2), (t_2, s_1), (t_2, s_2), (t_3, s_3), (t_3, s_4)$ instead of the full Cartesian Product of the two input streams. Observe that the first four input pairs of process d are derived from the input (x_1, y_1) and the remaining two input pairs are derived from (x_2, y_2). Because the two inputs of process d originate in the same process a, we shall form the Cartesian Product over the portions of the input streams which are generated by the same output pair of process a, e.g., (t_1, t_2) and (s_1, s_2), or (t_3) and (s_3, s_4). In order to derive the correct input combination, we mark process a as a Sync generator and the outputs generated by process a are separated by a special Sync signal. The Sync signals are then propagated through processes b and c, and reach process d in both inputs. Finally process d detects the same Sync signals in both inputs and then forms the Cartesian Product over the input portions which are enclosed by the corresponding pair of Sync signals.

After the data flow graph has been constructed, we determine all the multiple paths in the graph and mark the starting nodes of those paths as Sync generators. Different Sync generators generate different Sync signals. A process that receives two or more inputs from different channels merges the input streams according to the Sync signals carried in each input stream. The Sync signals may be duplicated during the merge process when they are nested in other Syncs. In the above example, process a is a Sync generator, hence the output streams generated by process a should be $(S_a, x_1, S_a, x_2, \text{END})$ and $(S_a, y_1, S_a, y_2, \text{END})$ respectively, where S_a represents a Sync signal generated by process a and "END" represents a special signal indicating the end of the

stream. Likewise, the output streams of process b and process c should be $(S_a, t_1, t_2, S_a, t_3, \text{END})$ and $(S_a, s_1, s_2, S_a, s_3, s_4, \text{END})$ respectively. Therefore, the input combination of process d becomes $(S_a, (t_1, s_1), (t_1, s_2), (t_2, s_1), (t_2, s_2), S_a, (t_3, s_3), (t_3, s_4), \text{END})$. Once a Sync signal is generated, it is propagated to (may be duplicated in) the other sibling AND processes through the communication channels in the data flow graph. The Sync signals will be removed at the father OR process before the output streams are sent out to higher level AND processes. Therefore, the Sync signals are local to the OR process and its AND descendants.

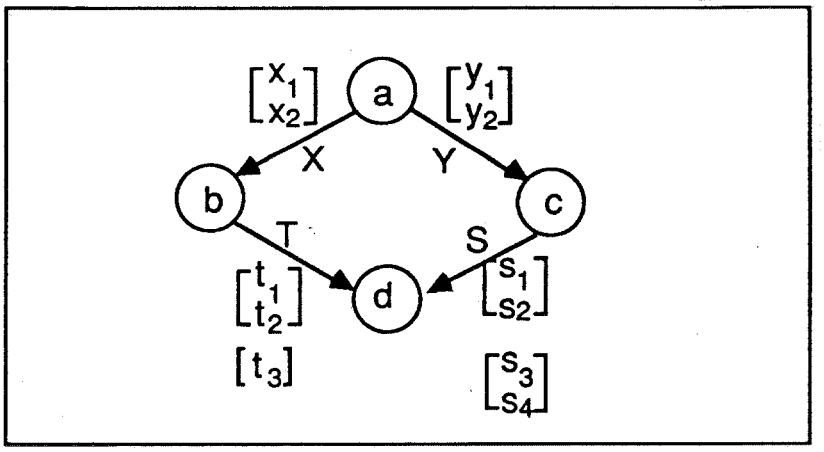

Figure 1. An Example with Multiple Path

2.2.2 Partially Instantiated Terms

When the producer of a variable binds the variable to a partially instantiated term, i.e., a term containing another variable, binding conflict may occur if that variable has more than one consumer. We solve this problem by adding so-called "dynamic links" into the graph to enforce the "one producer per variable" rule to the newly generated variable.

The data flow graph needs to be changed in two cases: (1) when a variable is bound to a partially instantiated term and this variable has more than one consumer, and (2) when two or more variables are bound to some terms containing the same variable. In both cases, one of the consumers of these variables is selected as the producer of the new variable and the dynamic links are directed from the new producer to all the rest of the consumers. The information about dynamic links is not provided during the construction of the data flow graph. Instead, such information is generated and sent to the selected producer of the new variable when an AND process binds some output variables to partially instantiated terms. A simple test on the binding values of all the output variables to test the above two cases is sufficient to determine whether dynamic links are

needed and how they are directed. Such a test is similar to DeGroot's type checking [3], except that we do the same check in every AND process without consulting the complex graph expression proposed by DeGroot.

The creation of dynamic links may introduce new Sync generators. The only process which may become a Sync generator is the producer of the new variable, which becomes a Sync generator when the node that binds a variable to the partially instantiated term is also a Sync generator. Those Sync Generators are identified and marked after the dynamic links are created. For more detail about dynamic links, see [6].

2.2.3 The AND process and the OR process

We briefly summarize the major tasks performed by an AND or an OR process. For full detail of the Sync Model, see [6].

AND process

- Call a merge algorithm to merge the input streams and bind the merged inputs to the input variables of the goal one at a time if the goal contains input variables.
- Perform type checking on the merged input and create dynamic links if necessary.
- Spawn OR processes and collect the results for each of the goals with bound input variables.
- Generate Sync signals to separate each of its outputs if it is a Sync generator.

OR process

- Unify the goal with a given clause.
- Return the bindings derived in the unification followed by an "END" to its father if the given clause is a unit clause.
- Construct the data flow graph of the guard literals and spawn AND processes for the guard if the given clause has a nonempty guard.
- Construct the data flow graph of the literals in the body and spawn AND processes for the body if all the AND processes for the guard return true or the given clause has an empty guard.
- Merge the partial solutions received from its descendants, remove the Sync signals and send the results to its father.

3 The Ordering Algorithm

By imposing that each shared variable has exactly one producer, we eliminate binding conflicts. To construct the data flow graph of AND literals, an Ordering Algorithm is applied in each OR process. The data flow graph is represented in two ways: by variable annotations in the literals and by a channel table containing the producer and consumer information of shared variables.

The Ordering Algorithm is performed in an OR process to construct the data flow graph of the AND literals after unification succeeds and the variables in the clause body are replaced by their binding values if they are instantiated after the unification. The Ordering Algorithm consists three major steps: (1) the construction of the data flow graph, (2) the refinement of the graph, and (3) the marking of the Sync generators. In the first step, variable annotations are used to determine the modes (input or output) of the uninstantiated variables in the AND literals. Initially, all the AND literals in the clause body are stored in an *Undecided Process List* (UPL). The algorithm determines the producer and the consumers of all the variables in the AND literals, adds annotations to all the variables, and then moves the literals to a *Fired Process List* (FPL). A Channel Table (CT) is also constructed to store the producer and consumers information of all the variables. Moreover, the literals are renumbered during this step so that their numerical order is consistent with their partial order in the data flow graph. In the second step, the data flow graph is further refined by creating "selective channels" and "True/False channels" for the literals that generate no output variables. As we shall see, this step is necessary to exploit the parallelism implied by the program so that a more efficient data flow graph can be constructed. In the third step, the algorithm searches for all the multiple paths in the data flow graph. If a multiple path is found, the algorithm marks the starting node of the multiple path as a SYNC generator. The complete algorithm will be elaborated in the remainder of this section.

Data Structure:

The following data structures are used in the algorithm:

- UPL – a list of AND literal and identifier pairs that are not fired yet†.
- FPL – a list of fired AND literals with all their variable arguments annotated. Each entry in the list contains an AND literal with annotated arguments, a Sync attribute, and a number attached to the literal to enforce a total order.

† "A literal is fired" means that a literal is moved from UPL to FPL and all its variable arguments are annotated.

- CT – a table of triples (Var,Producer,Consumers-list), to record the producer and consumers of a variable.
- S – a stack containing distinct nodes belonging to the paths starting from one specific node.

Besides, the AND literals are initially identified 1 to N from left to right in the clause body with the goal of the current OR process numbered 0.

Algorithm:

Step 0: Initialization

CT:= $\emptyset$; FPL:= $\emptyset$;

UPL:= list of all literals.

Step 1: Construction of the data flow graph:

In this step, the producer and the consumers of each shared variable are chosen and the variables in each literal are annotated.

A literal can be fired iff (1) all its input variables have a producer and the producers are already fired, and (2) the total number of output variables, input variables, and constant arguments of this literal is at least one. The first condition assures that a producer of a shared variable is always fired before the consumers of this variable. The second condition implies that the threshold [14] of each literal is one. If none of the unfired literals satisfies the above conditions, the leftmost unfired literal in the clause body is chosen to be fired next.

a. forall v_i: $v_i \in$ uninstantiated variables in the goal
add $\langle v_i, [], [0] \rangle$ into CT;

b. forall l: $l \in$UPL
do forall v_i: $v_i \in$ variable arguments in l
do if $v_i \notin$ CT $\rightarrow$ if v_i is output annotated $\rightarrow$ add $\langle v_i, l, [] \rangle$ into CT
| otherwise $\rightarrow$ add $\langle v_i, [], [] \rangle$ into CT
fi
| $v_i \in$ CT $\rightarrow$ if v_i is output annotated $\rightarrow$ CT.v_i.*producer* := l
| otherwise $\rightarrow$ skip
fi
fi
od
od ;

c. $index := 1$;
while UPL$\neq \emptyset$
do $fired$:=false;
forall l: $l \in$UPL
do forall v_i: v_i is unannotated $\wedge$ CT.v_i.*producer* $\neq$[]
mark v_i as an input variable in UPL;
b :=true;
forall v_i: v_i is an input variable in l
do x :=CT.v_i.*producer*;
$b := b \wedge (x \neq [] \wedge x > N)$
od {$b = \forall v_i$: v_i *is an input variable in l:* v_i *has a producer and the producer is fired*}
if b$\wedge$(#constant arguments+#output variables+#input variables>0)$\rightarrow$
{*beginning of firing process*}
newid := *index* + N;
forall v_i: $v_i \in$ *variableargumentsinl*
do if v_i is input $\rightarrow$ add *newid* into CT.v_i.*consumer*
|v_i is unannotated$\vee v_i$ is output $\rightarrow$ CT.v_i.*producer* := *newid*;
mark v_i as an output variable in UPL
|otherwise $\rightarrow$ skip
fi;
od
UPL:=UPL$-l$;
FPL[*index*] := l;
index := *index* + 1;
$fired$:=true
{*end of firing process*}
|otherwise$\rightarrow$ skip
fi
od ;

d. if $\neg fired \rightarrow l$:=UPL[1];
do "firing process"
|otherwise$\rightarrow$skip
fi
od ;

e. forall v_i: $v_i \in$ CT
do if CT.v_i.*consumer* = [] $\rightarrow$ CT:=CT$-v_i$
|otherwise $\rightarrow$ skip
fi
od .

Step 2: Refinement of the Graph

If some literals have no output variables in the data flow graph constructed in step 1, extra links need to be added into the graph to make sure the true/false results of this kind of literals will be transmitted to the goal.

Let's assume p is such a literal and X is an input variable of p. In this step, we first attempt to add so-called *selective channels* from p to the rest of the consumer literals of X. These channels transmit only the values of X that make p true. Meanwhile, the links between the producer of X to the consumers of X except p are removed from the graph.. If no selective channel is constructed for p, a *True/False channel* is added from p to the goal to transmit the results of p.

The insertion of selective channels should not cause cycles in the graph. To assure the acyclicity of the graph, we only add the selective channels such that the receiver of the channel is fired after all the antecedents of the sender. The antecedents of a literal are the producers of all the input variables of the literal.

forall l: $l \in$ FPL $\wedge$ l has no output variables
do $new := false$;
$prod := \emptyset$;
forall v_i: v_i is an input variable of l
add CT.v_i.*producer* into *prod*;
forall v_i: v_i is an input variable of l
do $c := CT.v_i.consumer$;
$cl := \{c_i | c_i \in c : (\forall p_j \in prod : c_i > p_j) \wedge c_i \neq l\}$;
if $l \in c \wedge cl \neq \emptyset \rightarrow$ add $\langle \overline{v_i}, l, cl \rangle$ into CT;
CT.v_i.*consumer* $:= c - cl$;
new :=true
|otherwise $\rightarrow$ skip
if
od ;
if $\neg new \rightarrow$ add $\langle t/f, l, [0] \rangle$ into CT
|otherwise $\rightarrow$ skip
fi
od .

Step 3: Marking of the Sync generators (Detection of the multiple paths):

A stack is built for each literal l in FPL that has more than one output channel. The descendants of l are pushed into the stack if they are not yet in the stack. This pushing process

continues until either all the descendants of l are in the stack or a descendant to be added to the stack is found to be already in the stack. In the second case, l is marked as a SYNC generator.

```
forall l : l ∈FPL ∧ #consumers(l) > 1
do    pt := 1; S := [l];
      while S[pt] ≠ ∅
      do    p := S[pt];
            forall v_i : v_i is a variable in p
            do    if CT.c_i.producer = p →
                        forall c_i : c_i ∈ CT.v_i.consumer
                        do if c_i ∉ S → push c_i into S
                           |c_i ∈ S → set Sync attribute of p to true in FPL; stop
                           fi
                        od
                  |otherwise → skip
                  fi;
            od
            pt := pt + 1
      od
od
```

The above algorithm always generates an acyclic data flow graph with one producer per shared variable. The ordering algorithm is correct for the following reasons:

1. The ordering algorithm selects exactly one producer for each variable.

2. The data flow graph generated in Step 1 is acyclic because a literal can be fired only when all the producers of its input variables have been fired (b is true in Step 1.c). Therefore, the producer of a given variable is always fired before all the consumers of that variable. The firing order of the literals implies their partial order in the data flow graph, thus, the graph has no cycles.

3. The refined data flow graph generated in Step 2 is acyclic because the redirected links do not create cycles in the refined graph. If a cycle were found in the refined graph, it would contain at least one redirected link, say (l_i, l_j). Let the cycle be $(l_i, l_j, \ldots, l_k, l_i)$, then l_k is the producer of one input variable of l_i and $l_j > l_k$ because a path exists from l_j to l_k. In Step 2, such a link (l_i, l_j) is never generated because l_j is excluded from cl. Therefore, the refined graph is also acyclic.

An Example

Figure 2 is a query:"Is there a student such that a professor teaches him two different courses in the same room?" for a data base of *S*tudents who take *C*ourses ($student(S,C)$), *P*rofessors who teach *C*ourses ($professor(P,C)$), and *C*ourses held on certain week*D*ays and *R*ooms ($course(C,D,R)$), [10]. To save space, the database of relations *student*, *course*, and *professor* are omitted here.

```
query(S,P):- student(S,C1),          (1)
             course(C1,D1,R),        (2)
             professor(P,C1),        (3)
             student(S,C2),          (4)
             C1≠C2,                  (5)
             course(C2,D2,R),        (6)
             professor(P,C2).        (7)
```

Figure 2. A Query for a database of students

To answer the query "$:- query(S,P)$.", we construct a process tree and map the initial goal to the root. In the OR process that is spawned by the root, we shall apply the ordering algorithm against the seven AND literals in Figure 2.

Since none of the variables are annotated in the definition of *query*, we select the producers of the shared variables by imposing the left-to-right order of the literals and as a consequence, the data flow graph constructed by Step 1 is shown in Figure 3.

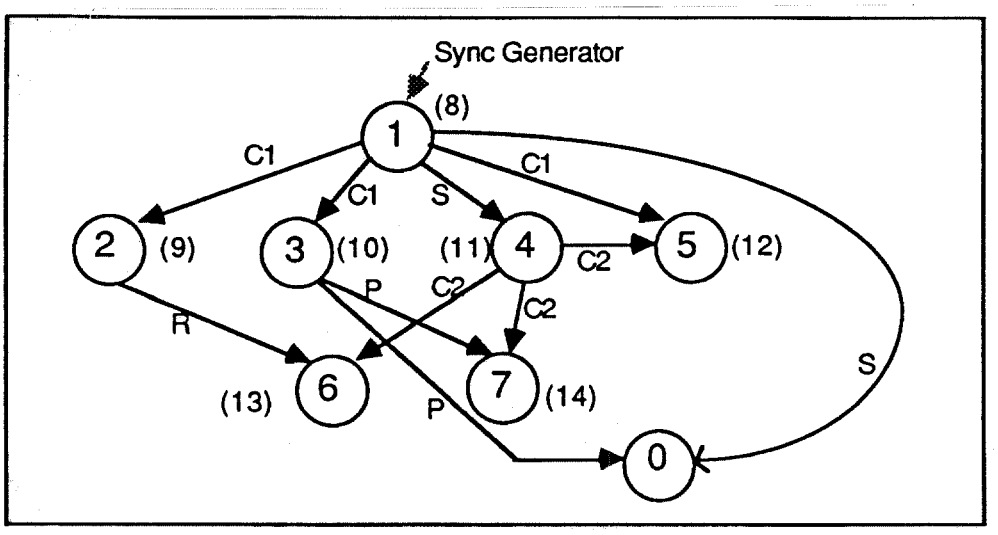

Figure 3. The Data flow graph of $query(S,P)$

In Step 1, the literals are renumbered so that their numerical order implies their partial ordering in the graph. The new identifiers of the literals are enclosed in the parentheses next to each node

in Figure 3. Notice that literals (5), (6), and (7) don't generate any outputs. After adding selective channels to these literals by Step 2, the refined data flow graph is shown in Figure 4. Comparing with the previous graph, we found that the variable $C2$ is transmitted sequentially from literal (4) to (5), (6) and (7) in the refined graph instead of transmitted in parallel in the original graph. At first glance, the refined graph seems to have less parallelism than the original one. In fact, the latter one is more efficient than the former one because literal (6) or (7) only receives the values of $C2$ such that literal (5) or (6) is proved true. Therefore, the values of $C2$ generated by (4) will first be filtered by (5), then sent to (6) and so forth. Unnecessary computations are avoided in (6) and (7) because invalid values of $C2$ won't be received by them. Also notice that no selective channels are constructed for $C1$ at literal (5) because the consumers of $C1$, (2) and (3), are both fired before the producer of $C2$, i.e., (4). To assure the acyclicity of the graph, the channels for $C1$ remain unchanged.

In Step 3, a stack is built up for literal (1). A multiple path is found when (5) is going to be pushed into the stack twice. Therefore, (1) is marked as a Sync generator. No more stack is needed because all the other literals have exactly one descendant each.

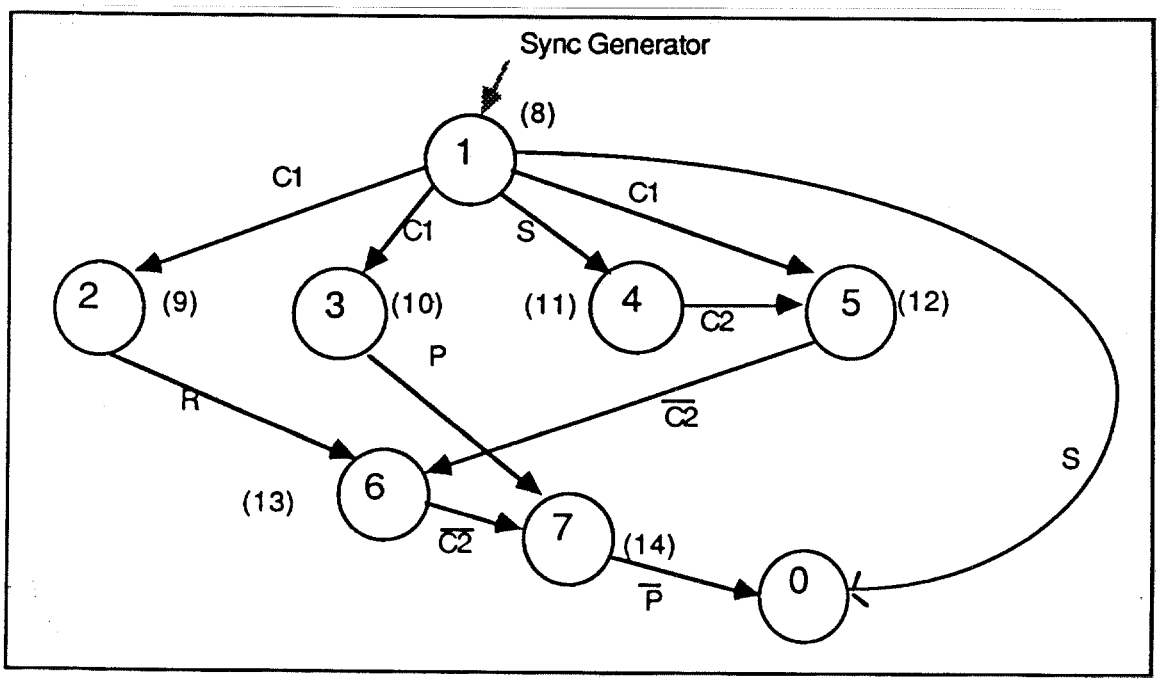

Figure 4. The Refined Data flow graph of $query(S, P)$

The average complexity of the ordering algorithm is $O(n \lg n)$ with n AND literals. In most cases, the AND literals in a clause body are almost-ordered, therefore, a linear complexity can be

achieved. For detail analysis of the complexity of the ordering algorithm, please see [6].

4 The Merge Algorithm

In the Sync Model, a process has to handle multiple input streams from different sources. For example, an OR process has to merge all the partial solutions from its AND descendants to form the solutions of this OR process, and an AND process needs to merge the input streams from other sibling AND processes to form input combinations to itself. It is particularly true for a nondeterministic program, in which multiple partial solutions may be generated, transmitted and validated by different processes. A merge algorithm that synchronizes the execution of all the cooperating processes is the crucial part of our Sync Model.

The merge algorithm in an AND process is basically the same as the one in an OR process. The only difference is that the input stream of the latter one may contain True/False values instead of variable bindings. In the following, the merge algorithm refers to the one in an AND process.

The merge algorithm operates only when there exist two or more input variables in a process. An input stream consists of SYNC signals, variable bindings, and an END signal at the end. A variable binding is a pair consisting of a variable name and its binding value. The SYNC signal carries the process identifier that identifies the generator of the Sync signal. SYNC signals are nested when the receiving node belongs to two or more different multiple paths. In essence, the merge algorithm forms a Cartesian Product over the input streams to form all the possible input combinations. When SYNC signals appear, the algorithm forms Cartesian Product over part of the input streams separated by pairs of identical SYNC signals. In other words, only the input elements in between the corresponding pair of SYNC signals can be combined and the input streams are thus synchronized by the SYNC signals.

In the rest of this section, the base-case algorithm (i.e., no input stream contains SYNC signals) is described in the next subsection. The Cartesian Product implemented as nested loops is inefficient because the process may keep waiting for the inputs from a slow channel. A more efficient algorithm is given in Figure 5. This algorithm reduces the waiting time by forming the Cartesian Product over the available portions of input streams while the rest of the inputs are not there yet. The general algorithm with input streams containing SYNC signals is presented in Section 4.2. Figure 6 is the general algorithm for two streams. The general algorithm is a recursive algorithm which recursively peels off SYNC signals in two streams and finally forms the Cartesian Product over the data inputs

enclosed by the innermost SYNC signal pair with the base-case algorithm. The algorithm for n streams can be derived by generalizing the two-stream algorithm. In the last section, a correctness proof for the n-input general merge algorithm is presented.

Throughout the algorithms, $buf[i,j]$ is used to represent the j-th input in the i-th input buffer, where $1 \leq i \leq n$ and n is the total number of input buffers. Each buffer is assumed to have enough capacity to store the whole input stream. The $index[i]$ points to the position which is currently being merged and $avail[i]$ points to the top of the available portion of buffer i. Procedure $put(entry)$ adds a new element *entry* into the output queue, where *entry* can be a SYNC signal, an array of n input bindings or an "END" signal.

4.1 Base-case Algorithm

Since the merge algorithm is operating concurrently with the receiving of inputs in each input buffer, the simple iterative loop implementation may be inefficient due to waiting for the inputs from a slow channel. A more efficient implementation is shown in Figure 5.

This algorithm forms the CP (abbreviation for Cartesian Product) over the available portions of the n input streams repeatedly. Whenever an input buffer receives new inputs, Procedure *cp* is called repeatedly to form the CP over the newly received inputs and the available portions of the other input buffers. Then $avail[j]$ is advanced to the location of the newest available input. The algorithm repeats the above operations for each input buffer until the new input in all the input buffers is "END".

```
{ Global Variables}
integer n;  { number of input buffers}
integer array index[1:n], avail[1:n];  { pointers}
input buffer buf[1:n,1:m];  { n input buffers with length m which are large enough to contain
                              the whole input streams}
buffer entry[1:n]; { a buffer to contain the next output}

{ Cartesian Product of the available portions of the n input buffers except the i-th buffer
  which is fixed to an element e}
procedure  cp(e,i);
begin
    entry[i]:=e;
    cp1(i,1)
end.

{ Cartesian Product over the available portions of buf[k] to buf[n] except buf[i]}
procedure  cp1(i,k);
begin
```

```
    [ k>n → put(entry)
    | k=i → cp1(i,k+1)
    | otherwise → l:=1;
                  *[ l≤avail[k] → entry[k]:=buf[k,l];
                                  cp1(i,k+1);
                                  l:=l+1
                   ]
    ]
end.

{ Main Program }
begin
    i:=1;
   *[ i≤n → index[i]:=1; avail[i]:=0; i:=i+1 ];
    i:=1;
   *[ ∃k: 1≤k≤n: buf[k,index[k]]≠'END' →
        *[ i≤n → *[ ¬empty(buf[i,index[i]])∧buf[i,index[i]]≠'END' →
                                                  cp(buf[i,index[i]],i);
                                                  index[i]:=index[i]+1
                  ];
                  avail[i]:=index[i]-1;
                  i:=i+1
         ]
    ]
end.
```

Figure 5. Base-case Algorithm

4.2 General Algorithm

If SYNC signals appear in at least one input stream, the general merge algorithm applies. We first present the general algorithm for two input streams and later show how to generalize the algorithm to n input streams.

In the ordering algorithm, the literals have been renumbered so that their numerical order is compatible with their partial order in the data flow graph. The linear ordering of the Sync signals in an input stream is always assured by the merge operation which performs an n-way merge on n input streams.

The general algorithm consists of two principal operations: merge on the same Sync signals and merge on different Sync signals. First, let two input streams contain the same Syncs, say S, and the two input streams are $A = (S, A_1, S, A_2, \ldots, S, A_n, \text{END})$ and $B = (S, B_1, S, B_2, \ldots, S, B_n, \text{END})$, then the merge result is a sequence of CP's over the corresponding portions of the two input sequences which are separated by a pair of consecutive S's, i.e.,

$$A \times B = (S, A_1 \times B_1, S, A_2 \times B_2, \ldots, S, A_n \times B_n, \text{END}) \tag{1}$$

where A_j stands for a sequence of data inputs, so as B_j for $1 \leq j \leq n$, and $A_j \times B_j$ stands for the CP of A_j and B_j.

The second principal operation handles the merge of two sequences with different Syncs. Let two input streams be $A = (S1, A_1, S1, A_2, \ldots, S1, A_n, \text{END})$ and $B = (S2, B_1, S2, B_2, \ldots, S2, B_m, \text{END})$, and let S1<S2 so that S1 becomes the outer Sync in the merge result. The linear ordering of the Sync signals in a merged stream guarantees that the common Syncs appearing in two input streams are in the same order, therefore, the merge algorithm functions correctly. The merge result can be computed as follows:

$$\begin{aligned} A \times B &= (S1, A_1 \times B, S1, A_2 \times B, \ldots, S1, A_n \times B, \text{END}) \\ &= (S1, S2, A_1 \times B_1, S2, A_1 \times B_2, \ldots, S2, A_1 \times B_m, \\ &\quad S1, S2, A_2 \times B_1, S2, \ldots, A_2 \times B_m, \\ &\quad S1, S2, \ldots\ldots, S2, A_n \times B_m, \text{END}) \end{aligned} \tag{2}$$

The merge result is actually the CP of all the data inputs of the two streams when the two input streams contain different Syncs. In order to maintain the synchronization information, we first do the CP's over the whole input stream B and a portion of stream A, i.e. A_i for all i and separate the CP's by S1. In each $A_i \times B$, again we do a set of CP's of $A_i \times B_j$ for all j and separate them by S2. The CP "$A_i \times B_j$" contains no Sync signals, hence the base-case algorithm can be applied. In the result, the number of Sync signals S1 is preserved, i.e., n, and the number of Sync signal S2 is increased to $n \times m$ because S2 is nested inside S1.

The general algorithm for two input streams is recursively defined on the two principal operations. The Sync sequences of the input streams are linearly ordered, i.e., a Sync signal is larger than all the Syncs which are outer to it and smaller than all the Syncs inner to it. In each recursion, the outermost Sync signals of the two input streams are checked. If they are the same, the first principal operation is called. If they are different, the second principal operation is called. The merge algorithm is called recursively to compute each $A_i \times B_i$ in (1) or each $A_i \times B$ in (2). When the merge algorithm is called to merge two input streams without any Sync signals, the base-case algorithm is applied to get the CP. The merge result preserves the linear ordering of the Sync sequence. Figure 6 presents the major procedures of the merge algorithm: *merge*, and *scanto*. Procedure *merge* merges the input streams in $buf1$ and $buf2$, and puts the result in an output queue. Boolean function *sync* checks whether the given argument is a Sync signal or not. Procedure *merge* has a guarded command with four alternatives: (1) neither of the inputs contains

Sync signals: *cp* is called to derive the Cartesian Product, (2) either $buf1$ contains Sync signals and $buf2$ does not, or both inputs have Sync signals and the outermost Sync of $buf1$ is smaller than that of $buf2$: the second principal operation applies, (3) same condition as (2) with $buf1$ and $buf2$ switched: the second principal operation also applies with A and B switched, and (4) both inputs contain Sync signals and the outermost Syncs of the two inputs are the same: the first principal operation applies. Procedure *scanto* divides the input buffer into two parts by the first occurrence of some specific SYNC signal S. Procedure *cp* is the base-case merge algorithm which generates the CP of the data elements in two buffers.

```
procedure  merge(buf1,buf2)
begin
    [buf1=∅∨buf1="END"∨buf2=∅∨buf2="END" → skip
    |otherwise→ A:=buf1[1]; B:=buf2[1];
        [¬sync(A)∧¬sync(B) → cp(buf1,buf2)                          (1)
        |sync(A)∧(¬sync(B)∨(A<B)) → scanto(buf1,A,buf11,buf12);     (2)
                                    put(A);
                                    merge(buf11,buf2);
                                    merge(buf12,buf2)
        |sync(B)∧(¬sync(A)∨(B<A)) → scanto(buf2,B,buf21,buf22);     (3)
                                    put(B);
                                    merge(buf1,buf12);
                                    merge(buf1,buf22)
        |sync(A)∧sync(B)∧(A=B) → scanto(buf1,A,buf11,buf12)         (4)
                                  scanto(buf2,B,buf21,buf22);
                                  put(A);
                                  merge(buf11,buf12);
                                  merge(buf12,buf22)
       ]
    ]
end of procedure merge.

procedure  scanto(buf,S, buf1,buf2)
begin
     i:=2;
    *[ buf[i]≠S∧buf[i]≠"END" → buf1[i]:=buf[i]; i:=i+1];
     j:=1; N:=length(buf);
    *[ i≤N → buf2[j]:=buf[i]; i:=i+1; j:=j+1]
end of procedure scanto.
```

Figure 6. General Algorithm for two buffers

If there are more than two input buffers and some of them have one or more SYNC signals, the above algorithms can be generalized easily. With n input streams, in which each has an ordered Sync sequence, the merge algorithm applies recursively to remove the smallest Sync signal of the n outermost ones of the input streams one at a time. When the smallest Sync is common to several input streams, all those Syncs will be removed at once. When none of the input streams

contains Sync signals, the Cartesian Product over n input streams is performed. For instance, if $merge(buf1, buf2, \ldots, bufn)$ is called and a smallest Sync S is found in both $bufi$ and $bufj$, the following program is executed:

```
scanto(bufi,S,bufi1,bufi2);
scanto(bufj,S,bufj1,bufj2);
put(S);
merge(buf1,...,bufi1,...,bufj1,...,bufn);
merge(buf1,...,bufi2,...,bufj2,...,bufn);
```

The merge algorithm in an OR process merges the partial solutions received from its AND descendants to form all the legal solutions of this OR subtree. The partial solutions received from one AND descendant could be variable bindings or true/false values. The true/false values are used to select the merge result from other channels. If the value is true, the merge algorithm merges the partial solutions as usual. If the value is false, the merge algorithm skips the merge operation and returns false instead. In addition, the merge algorithm in an OR process eliminates all the Sync signals in the merge result so that the solution stream sent up to the father AND process contains no Sync signals.

4.3 Correctness Proof

In order to prove that the merge algorithm produces all the correct combinations of multiple inputs of a process, we shall define the syntactic structure of an "input stream" and give a formal treatment of how an AND process transforms one or more input streams into an output stream.

Definition: An *input stream* $\Sigma_R(D)$ can be defined recursively:

1. $\Sigma_\emptyset(D) \equiv D$

2. $\Sigma_{R\cup\{i\}}(D) \equiv \Sigma_R(\Sigma_{\{i\}}(D)) \equiv \Sigma_R((S_i, D_v)^{n_i}), \forall j \in R : i > j.$

where R is an ordered set of integers. Each element in R is a Sync that appears in the input stream. Let's call R the *Sync sequence* of this input stream. We slightly abuse notations and represent R by the array $R[i]$, s.t., $i < j \Rightarrow R[i] < R[j]$. Σ_R is an operator defined recursively over the input data, D, where D is the input stream with all the Syncs removed. Applying $\Sigma_{\{i\}}$ over D is to divide D into n_i groups and separate each group by a Sync S_i. Each group of input data, D_v, is called a *data segment*, which is uniquely identified by a vector, v. In (2), v is a vector of length $(r+1)$ where $|R| = r$ and $v[r+1] = k$ for $1 \leq k \leq n_i$. Therefore, D_v represents a data segment that is

produced by the k-th output of the Sync generator S_i. Besides, $(S_i, D_v)^{n_i}$ is a regular expression denoting the concatenation of the string (S_i, D_v) n_i times. Notice that the data segment D_v is changed every time the syntactic structure of the input stream is transformed. The above notation is used to represent the syntactic structure of an input stream. How D_v is changed by different transformations of the input stream will be explained later.

There are two ways of changing the structure of an input stream in our model. First, if an AND process is a Sync generator, the structure of the output stream is derived by concatenating an extra Sync signal to the Sync sequence of the input stream. Second, if an AND process has several inputs, say n, the structure of the merge output can be derived by an n-way merge of the n Sync sequences. Figure 7 shows the two possible transformations of an AND process given one input and one output. In Figure 7.a, the structures of the input and the output streams are the same because the AND process is not a Sync generator. In Figure 7.b, the AND process is a Sync generator which generates Sync S_i and the output stream has the structure $\Sigma_{R\cup\{i\}}$. Because of the total ordering of the Sync generators, i is guaranteed to be larger than any element in R. Figure 8 shows the input-output transformation of the merge algorithm, given n input streams. The Sync sequence of the output is derived by n-way merge of the n input Sync sequences. An AND process with n inputs and one output can be represented by one merge operation (Figure 8) followed by one of the two AND operations (Figure 7) depending on whether the AND process is a Sync generator or not.

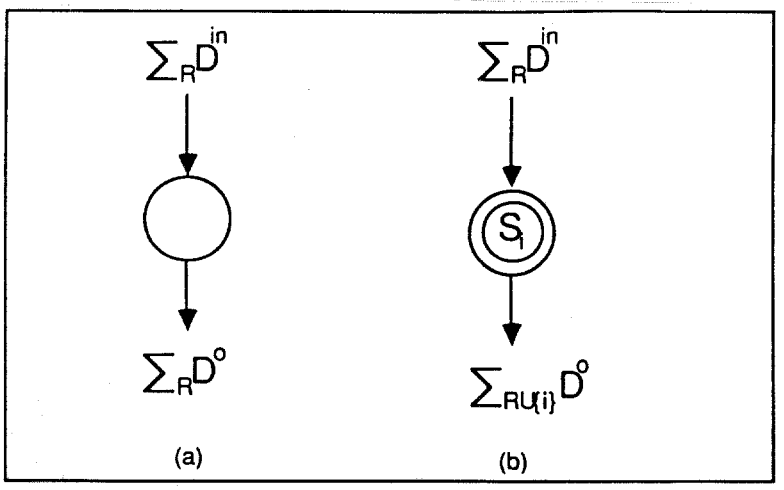

Figure 7. The transformation of an AND process with single input

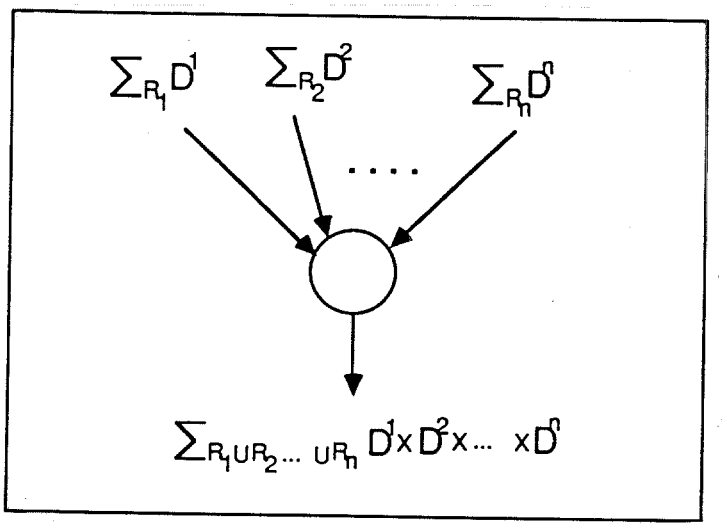

Figure 8. The transformation of the merge algorithm with two inputs

The data segments in the input stream are changed differently in the two transformations described above. Since we are only interested in the merge result, we'll only consider the second case, i.e., the transformation due to the merging of two input streams.

Definition: An *ordered union* operator "$\sqcup$" is defined as $R = R_1 \sqcup R_2$, where R, R_1 and R_2 are ordered sets (i.e., the elements in the set are sorted in ascending order) and $R = R_1 \cup R_2$. In other words, it is equivalent to a two-way merge.

Definition: An *ordered join* operator "$\underline{\sqcup}$" is defined as $v_R = v_{R_1} \underline{\sqcup} v_{R_2}$, where $R = R_1 \sqcup R_2$, v_R, v_{R_1} and v_{R_2} are vectors with length $|R|$, $|R_1|$ and $|R_2|$ respectively. v_R is the result of joining v_{R_1} and v_{R_2} on the common elements of R_1 and R_2. More precisely, $v_R = v_{R_1} \underline{\sqcup} v_{R_2}$ iff

1. $\forall i, j : R_1[i] = R_2[j] \Rightarrow v_{R_1}[i] = v_{R_2}[j]$ and
2. $v_R[i] = \begin{cases} v_{R_1}[j], & \text{if } R[i] = R_1[j]; \\ v_{R_2}[k], & \text{if } R[i] = R_2[k]. \end{cases}$

Theorem 1. *Given two input streams $\Sigma_{R_A}(D^a)$ and $\Sigma_{R_B}(D^b)$, the result generated by the merge algorithm is $\Sigma_{R_C} D^c$, where $R_C = R_A \sqcup R_B$. Moreover, D^c is defined as the Cartesian Product of D^a and D^b such that*

$$D^c_{v_c} = D^a_{v_a} \times D^b_{v_b} \qquad \text{with} \quad v_c = v_a \underline{\sqcup} v_b \tag{3}$$

Proof: Let the length of R_A and R_B be t_a and t_b respectively. This theorem can be proved by induction on the ordered pair (t_a, t_b), where $(t_a, t_b) < (t'_a, t'_b)$ iff $t_a < t'_a$, or $t_a = t'_a$ and $t_b < t'_b$.

It is easy to derive the proof from the program in Figure 6. The complete proof is given in [6]. ∎

From Theorem 1, we derive the merge result with two arbitrary input streams. The remaining task is to show that the merge result is correct. Given a process with two inputs, a legal input combination is an input pair such that the input elements of the pair are originated from the same output of a common ancestor along the two input paths. An input path is a path containing the current process, one of the two input links, and tracing back to any ancestor of the current process. There are many such paths. If a process is shared by any two input paths, in which each contains one different input link, then only the inputs which are derived by the same output of that process can be combined. Notice that such a common ancestor is marked as a Sync generator. Therefore, by observing the Sync sequences of the two input streams, we can determine all the common ancestors which affect the merge result along the two input paths.

Theorem 2 shows that the merge result in Theorem 1 indeed contains all the legal input combinations.

Theorem 2. *The result of the merge algorithm contains all the legal input combinations.*

Proof: Supposed that the two input streams in Theorem 1 have n common Sync signals, i.e., $|R_A \cap R_B| = n$, we need to prove that all the inputs that are derived from the same outputs generated by the n Sync generators are combined. Let P_i be the Sync generator that generates a Sync signal S_i. Then, each output generated by P_i is separated by a pair of S_i's. By propagating the output stream of P_i throughout the data flow graph, the syntactic structure of the output stream may or may not be changed. If the syntactic structure of the output stream is not changed, any result derived by the k-th output of P_i is appeared in the same data segment enclosed by the corresponding pair of S_i's. When the syntactic structure of the output stream is changed by merge operations or the generation of new Syncs, S_i may be further nested into other Sync signals. In this case, the results derived by the k-th output of P_i are divided into several data segments and spread into different locations. Generally speaking, with the input stream A in Theorem 1, the inputs that are derived by the k-th output of process P_i are the union of all the data segments with the j-th element of its id vector being k, where j is the position that S_i is placed in the Sync

sequence R_A.

$$\bigcup_{\substack{\forall v_a: \\ v_a[j]=k \wedge R_A[j]=i}} D^a_{v_a}$$

where $\bigcup_{\forall v_a: v_a[j]=k \wedge R_A[j]=i}$ is used as an abbreviation for a sequence of unions with the index v_a satisfing the condition specified in the subscript of $\bigcup$.

Assume there are n common Syncs, $S_{i_1}, S_{i_2}, \ldots, S_{i_n}$, in the two input streams. We will show that the merge result in the case the Sync generator P_{i_j} generating the t_j-th output, for all j, $1 \leq j \leq n$, is the Cartesian Product of the portions of the two input streams under the same condition. Let k_j, l_j and m_j be the locations where S_{i_j} appears in R_a, R_B and R_C, i.e., $R_A[k_j] = R_B[l_j] = R_C[m_j] = i_j$, for $1 \leq j \leq n$. Then the above relation can be formulated as follows:

$$\bigcup_{\substack{\forall v_c: \\ (\forall j:1\leq j\leq n: \\ v_c[m_j]=t_j)}} D^c_{v_c} = \bigcup_{\substack{\forall v_a: \\ (\forall j:1\leq j\leq n: \\ v_a[k_j]=t_j)}} D^a_{v_a} \times \bigcup_{\substack{\forall v_b: \\ (\forall j:1\leq j\leq n: \\ v_b[l_j]=t_j)}} D^b_{v_b}. \tag{5}$$

Eq. (4) can be derived from Eq. (3) easily. First add a big union $\bigcup_{\forall v_c:(\forall j:1\leq j\leq n:v_c[m_j]=t_j)}$ to both sides of (3). Then divide the unions at the right hand side into two independent sets of unions and then move the unions inside the CP and associate the first set of unions to D^a and the second set of unions to D^b.

$$\begin{aligned}
\bigcup_{\substack{\forall v_c: \\ (\forall j:1\leq j\leq n: \\ v_c[m_j]=t_j)}} D^c_{v_c} &= \bigcup_{\substack{\forall v_c: \\ (\forall j:1\leq j\leq n: \\ v_c[m_j]=t_j)}} (D^a_{v_a} \times D^b_{v_b}) \\
&= \bigcup_{\substack{\forall (v_a \sqcup v_b): \\ (\forall j:1\leq j\leq n: \\ v_a[k_j]=v_b[l_j]=t_j)}} (D^a_{v_a} \times D^b_{v_b}) \\
&= \bigcup_{\substack{\forall v_a: \\ (\forall j:1\leq j\leq n: \\ v_a[k_j]=t_j)}} \bigcup_{\substack{\forall v_b: \\ (\forall j:1\leq j\leq n: \\ v_b[l_j]=t_j)}} (D^a_{v_a} \times D^b_{v_b}) \\
&= \bigcup_{\substack{\forall v_a: \\ (\forall j:1\leq j\leq n: \\ v_a[k_j]=t_j)}} D^a_{v_a} \times \bigcup_{\substack{\forall v_b: \\ (\forall j:1\leq j\leq n: \\ v_b[l_j]=t_j)}} D^b_{v_b}.
\end{aligned}$$

Therefore, we can conclude the merge algorithm gives all the legal input combinations. ∎

With the above theorems, we can show that the Sync Model is complete, i.e., the Sync Model generates all the solutions for a given program.

From Kowalski [4], we know that each successful computation of an initial goal can be represented as a subtree of the AND/OR tree, i.e., the process tree in our Model. Such a subtree

starts from the root, expands by including exactly one descendant OR process for each of its AND process and all the descendant AND processes for each of its OR process, and ends with leaf nodes that successfully terminates.

Since any successful computation can be mapped onto a subtree in the Sync Model, if we can prove that such subtree generates the same solution as this successful computation, then the Sync Model is proved to be complete.

Theorem 3. *The Sync Model is complete.*

Proof: We first prove that a subtree that represents a successful computation generates the same solution as this computation. Let's first choose any OR process in a subtree that corresponds to a successful computation. Assume this OR process contains a goal g and a clause "$g1 :- p_1, p_2, \ldots, p_n$.". Let $X_1, X_2, \ldots, X_m$ be the variables within this clause and the successful computation gives a unique solution to these variables, i.e., $t_1, t_2, \ldots, t_m$. Moreover, let each p_i contains a set of input variables and a set of output variables. The input-variable set and the output-variable set of any p_i are disjoint and both of them are subsets of $(X_1, \ldots, X_m)$.

Let's assume that the subtree under each p_i produces the correct solutions for the output variables of p_i if the input variables are bound to the correct values. Here, the correct solution of a variable X_i is meant to be t_i. Therefore, any process p_i that has no input variables will generate the correct solutions to its output variables. Furthermore, any p_i with nonempty input-variable set will produce the correct solutions to its output variables if the producers of its input variables generate the correct solutions. The above statement is obviously true if p_i has only one input variable. It is also true if p_i has more than one input variable because the merge algorithm in p_i always generates the correct input combinations from Theorem 2. Therefore, the OR process generates the correct solution for its goal g assuming the subtrees under each p_i are correct. Furthermore, if in the subtree corresponding to a successful computation, there is an OR process which contains a unit clause. This OR process is always a leaf node and it generates the correct solutions to the output variables of the goal in the process. Thus, by induction, the subtree corresponding to a successful computation will generate the correct solution for that computation.

From the other direction, we shall also prove that any minimal subtree which produces an answer corresponds to a successful computation. A minimal subtree is a subtree which contains no failure nodes. The proof is similar to the proof above and thus omitted here.

Since any successful computation can be mapped onto a subtree in the Sync Model and each subtree generates the correct solution for the corresponding computation, we conclude that the Sync Model generates all the solutions for a given program and therefore it is complete. ∎

5 Conclusion

We have presented a model for the parallel execution of logic programming on a message-passing multiprocessor system. AND parallelism is carried out by constructing an efficient data flow graph dynamically The mechanism that is used to synchronize the multiple partial solution flows in the data flow graph makes it possible to realize both AND parallelism and OR parallelism without any form of backtracking.

Our model is complete. It handles both deterministic and non-deterministic programs, and it is particularly good for non-deterministic programs with multiple solutions. It is able to handle a pure logic program as well as an extended logic program with variable annotations and guarded clauses.

In our model, the AND/OR tree is searched in both breadth-first and depth-first manner. Consider two sibling AND processes that share a common variable. The subtree under the producer of the variable will be searched first and then the search for the consumer and its subtree can be started. If the producer produces multiple solutions to the variable, the execution of the two sibling AND processes are pipelined. Although this approach seems to be less parallel than purely breadth-first search of the AND/OR tree, our model is in fact more efficient because we avoid unnecessary computations in the consumer process. In a purely breadth-first search, invalid bindings of the shared variable are sent to the consumer and later found invalid by a process in the subtree of the producer.

We believe that any form of backtracking – "naive" or "intelligent" – should be totally eliminated from an OR-parallel model of logic programming. Backtracking simply means complicated control and high overhead. The synchronization mechanism proposed in the Sync Model is clean and simple. Although we need extra Synchronization signals, we don't need to send the complete set of bindings and thus, the overhead is actually lower.

Our Model can be modified to handle stream parallelism as well. Extended with tail recursion optimization [6], our model becomes an efficient parallel model that exploits all kinds of parallelism

inherent in a logic program. The mapping from the Sync Model onto the Sneptree, which is chosen as the target machine for our Model, is found to have minimal mapping cost in terms of load balancing and communication overhead. Therefore, it is feasible to construct a message-passing multiprocess system based on the Sneptree architecture to implement the Sync Model effectively.

Acknowledgements

The authors would like to thank Kevin Van Horn for his valuable comments and the members of the "Thursday Meeting Group" for their carefully reading on an earlier draft.

Reference

[1] Chang, J.H., and D. DeGroot, *AND-Parallelism of Logic Programs Based on Static Data Dependency Analysis,* Dept. of Electrical Engineering and Computer Science, Univ. of Calif, Berkeley. Sep, 1984

[2] Conery, John S., *The AND/OR Process Model for Parallel Interpretation of Logic Programs,* Ph.D. Dissertation, TR204, University of California, Irvine, June 1983

[3] DeGroot, Doug, *Alternate Graph Expressions for Restricted AND-Parallelism.* Compcom 85, Spring, pp206-210, Feb. 1985.

[4] Kowalski, R.A., *Logic for Problem Solving,* Elsevier Nother Holland Inc., 1979.

[5] Li, P., and A.J. Martin, *The Sneptree – A Versatile Interconnection Network,* 5194:TR:85, Computer Science, Caltech, 1985.

[6] Li, P., *A Parallel Execution Model for Logic Programming,* Ph.D. Dissertation, Computer Science, Caltech, 1986.

[7] Lindstrom, G., *OR-Parallelism on Applicative Architectures,* Lab. for Computer Science, Mass. Institute of Tech., Jan, 1984

[8] Martin, A.J. and J. van de Snepscheut, *Networks of Machines for Distributed Recursive Computations,* TR:84:5147, Caltech, Computer Science, 1984

[9] Nakagawa, Hiroshi, *AND Parallel PROLOG with Divided Assertion Set,* 1984 International Symposium on Logic Programming, pp22-28, Feb, 1984.

[10] Pereira, L.M., and Porto, A., *Selective Backtracking for Logic Programs,* Departamento de Informatica, CIUNL no. 1/80 University Nova de Lisboa.

[11] Ramakrishnan, R. and A. Silberschatz, *Annotations for Distributed Programming in Logic,* TR-85-15, Department of Computer Science, University of Texas at Austin, 1985.

[12] Saraswat, V.A., *Problems with Concurrent Prolog,* CMU-CS-86-100, Department of Computer Science, Carnegie-Mellon University, 1985.

[13] Shapiro, Ehud Y., *A Subset of Concurrent Prolog and Its Interpreter,* TR-003, ICOT-Institute for New Generation Computer Technology, Jan, 1983, Japan.

[14] Wise, Michael J., *A Parallel PROLOG: the construction of a data driven model,* Conference Proceeding of the Symposium on LISP and Functional Programming, pp 56-66, ACM, August, 1982.